P2638

Intel response to P1915R1 std::simd (Issaquah 2023 – P2807R0)

Daniel Towner

Summary

Intel supports the std::simd proposal and thinks it is a welcome addition to C++

We have made some detailed suggestions for alterations and additions:

- P2638R1 – General comments
- P2663R1 – Proposal to add complex-value support
- P2664R1 – Proposal to add permutation support

This feedback is based on our experience and that of partners for writing network and signal processing code.

Intel has an example implementation of std::simd boost for gcc/llvm. We are using this to evaluate std::simd in real-world code.

Excellent support for arithmetic operators and functions, but a theme which will emerge is that std::simd should have better permutation operations.

Comments addressed in P1928R2

- New names for deduce, compatible, etc. Agreed.
- Default ABI tag shouldn't be `compatible'. Fixed by changing the default to native.
- Memory flags don't have a default – fixed to element_aligned.
- Simd_cast and implicit conversions can be confusing and inconsistent –fixed
- simd_mask didn't allow cast – Fixed by removing simd_cast.
- Generator missing for simd_mask constructor – Added.
- Use constexpr everywhere - Added

Issues addressed in P1928R3

- simd_mask reduction naming (e.g., popcount, count[lr]_[zero/one])
- Removed simd_mask::some_of
- Removed `where' and replaced with mask-overloaded/renamed functions and possible conditional operator.
- Added <bit> header for popcount, byteswap, count[lr]_[zero/one]
- Added conversion to and from std::bitset

Insert/extract

Proposal to add:

- `resize_simd_t<End-Begin>` `std::extract<Begin,End>(v)`
- `simd<>` `std::insert<Begin>(v, child)`
  - Returns new simd of compile-time size.
  - Original simd inputs are unchanged
  - Compile-time checking of boundaries.

Why?

Using **`split`** and **`concat`** is too verbose:

```
auto [t0, throwaway, t1] = split<Begin, End-Begin, Pad>(v);
updatedSimd = concat(t0, newData, t1);
```

It has tricky behaviour at the boundaries too.

Open questions:

- Names
- Runtime behaviour – next slide

Insert/extract open question

Should insert/extract allow run-time offset?

- `resize_simd_t<End-Begin>` `std::extract<_Size>(v, offset)`
- `simd<>` `std::insert(v, child, offset)`

Pros:

- General purpose interface potentially widens scope of use
- Compiler isn't prevented from using an efficient code sequence if the offset is known at compile-time, but it isn't guaranteed.

Cons:

- Intel's library doesn't have this and no-one has asked for it. Too general purpose for no reason?
- std::simd is a performance library. Introducing a potential inefficiency might be the wrong thing to do.
- The simd library is generally working with compile-time sizes (e.g., fixed_size, native_size, concat, split_by) – why are insert/extract different?
- Harder to handle boundary checking – would run-time checks be needed which throw exceptions, and does this impact generated code performance of what is a performance library?

Suggested polls:

Std::insert/extact or simd_insert, simd_extract?

Should insert/extract handle run-time offsets?

Direct resizing of simd

Proposal:

- Change the static element count of a simd or simd_mask:
  - `std::resize<N>(v, value=T())`

  Truncate to the new size or insert new supplied value to grow

Why?

- Resizing is a common operation in real code, particularly when interfacing to compiler builtins or intrinsics (for unusual instructions).

Open questions:

- Should we use an interface which matches that of vector, list ,etc, and allow silent truncation and insertion?
- Or, disallow a truncating resize and replace with extract instead to make it explicit that data is being removed. In that case, should this be called grow instead?

iota

Proposal:

- Add a function (or constant?) which returns a simd initialised with sequentially ascending values:
  - `simd<T,A>::iota()    // T(0), T(1), T(2), …`

Why?

- iota can be used to help build lookup-tables, or constants, especially when tied to constexpr:
  - `constexpr auto multiplesOf3 = mysimd::iota() * 3;`
- Alternatively a generator could be used, but is quite verbose for something that is common:
  - ```
    constexpr auto multiplesOf3 =
                simd<T,A>([](auto ix) { return idx * 3; });
    ```
    ```

Interleaved fused-multiply add/sub

Proposal:

- Allow explicit interleaved fused addition/subtraction of simd:
  - `fmaddsub(a, b, acc);  // Odds add, evens subtract`
  - `fmsubadd(a, b, acc);  // Evens add, odds subtract`

Why?

- No concise way to represent this in simd.
  - `auto r = conditional_operator(evenMask, fma(a, b, c), fma(, b, -c));`
- No need for fnmadd, fmsub, etc. Can be easily peepholed.
- Less need for this with complex support

P2663 – Support for complex simd

Summary

std::simd currently supports vectorisation of all arithmetic types, excluding bool.

We propose that complex types should also be permitted:

```
simd<std::complex<float>>
fixed_size_simd<std::complex<double>, 8>
```

This will map to native processor support where it exists in instruction sets (e.g., Intel AVX-512, ARM Helium).

We also propose to provide overloads to match the behaviour of std::complex API.

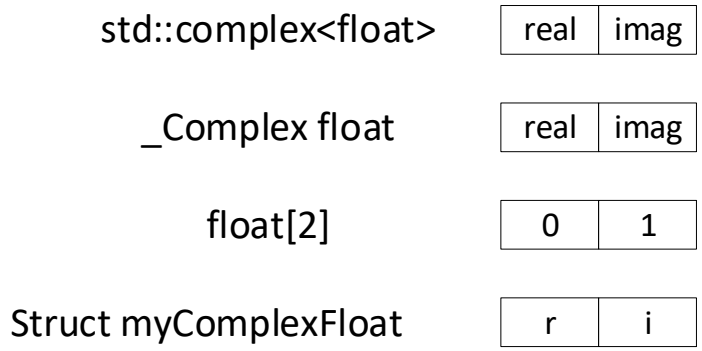
Storage of complex numbers

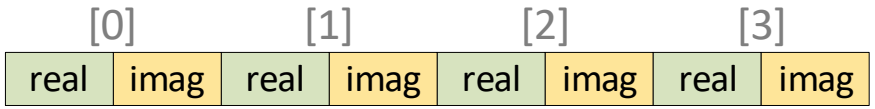
Complex numbers are pairs of real and imaginary values.

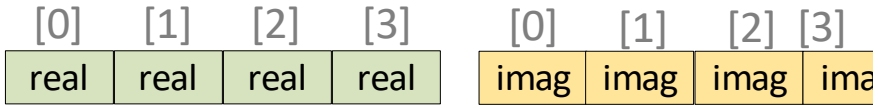
This format is used in many languages and software libraries, and is industry standard layout.

In memory or vector register storage each complex value is an atomic unit, so the real and imaginary elements are essentially interleaved.

Complex values could also be stored separately, which is equivalent to `std::complex<simd<float>>,` but that is beyond the scope of this proposal.

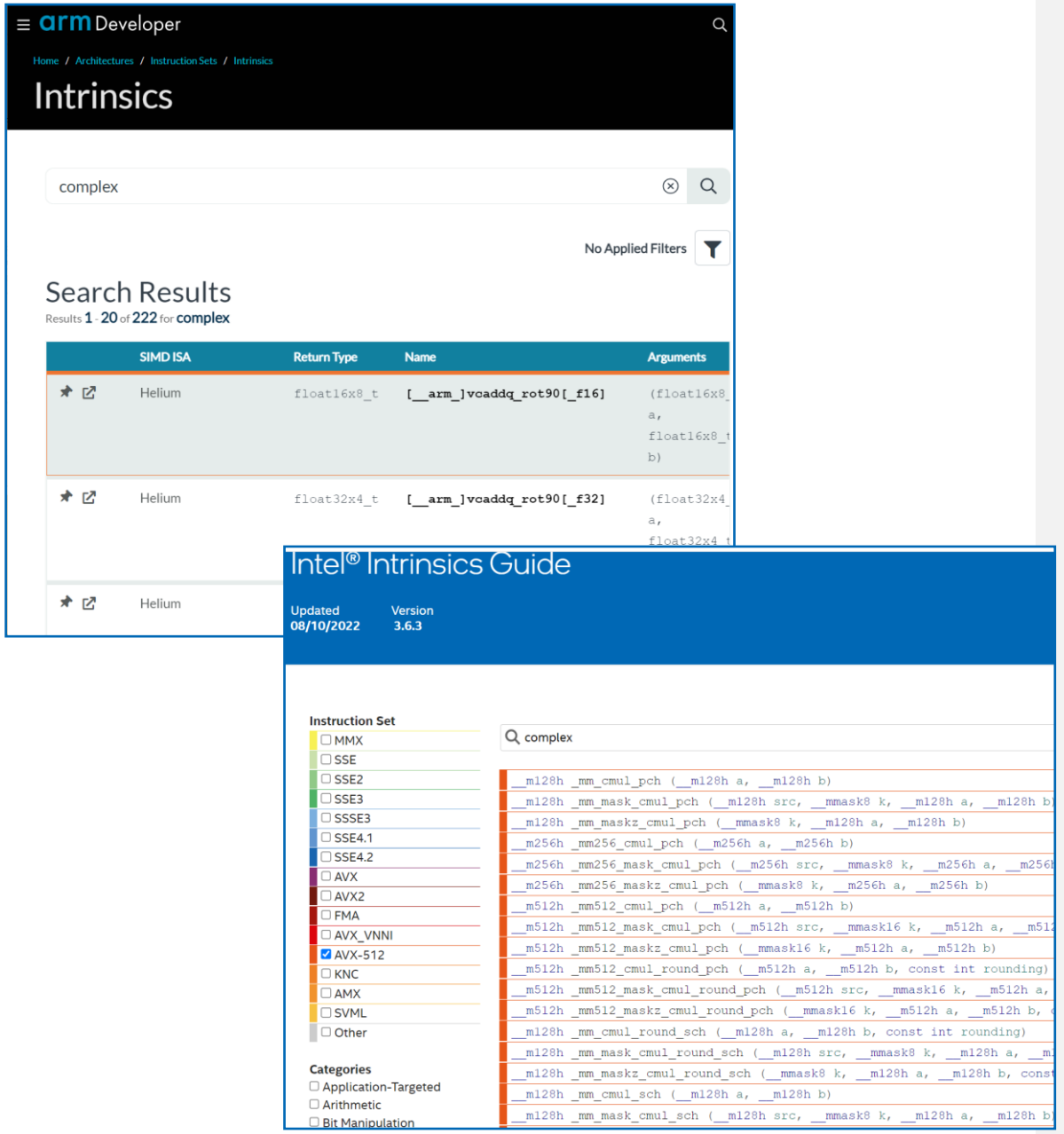






Implementation

- Both ARM and Intel have complex-valued vector instruction support. Other DSPs have support too.
- On targets which don't have native support, interleaved simd complex value can be almost entirely implemented in terms of the base simd implementation.



Main complex-simd proposal

Proposal:

- Allow `std::complex<T>` as a value type for `simd<>`
- Support all arithmetic and compound-assignment operators.
  - Operators like multiply and divide would do the per-element equivalent of their std::complex counterparts.
- All resize, split, concat, subscript, permute or other element access operations would work on complete std::complex<> granular elements.
- simd_mask of complex simd would have one mask bool per complex element.
- Overloads and operators which made no sense for complex values would be removed using concepts (e.g., relational operators like <, >=, etc..

```
constexpr friend mask_type operator<(const simd& lhs, const simd& rhs)
requires std::totally_ordered<_Tp>;
```

Why?

- Provide base support for simd values which allows easy access to the underlying hardware support where it exists.

Proposal to adopt complex API

Proposal:

- Adopt the API from std::complex<>
- Add complex methods to std::simd:
  - `simd<T, ABI> simd<std::complex<T>,ABI>::real()`
  - `void simd<std::complex<T>,ABI>::imag(simd<T, ABI> v)`
  - `simd<std::complex<T>,ABI>::conj()`
- Add maths function overloads:
  - sin/cos/log/exp/sqrt/etc
    - Return a simd<complex<T>>
  - arg/norm/abs
    - Return a simd<T> (i.e., real-valued simd with same number of elements)

Why?

Allow users to write generic code which works on either scalar or simd complex values interchangeably.